FEATURE ARTICLE

Kenneth Baker

Self-Tuning PD Algorithm for the 68HC11

The day-to-day changes in power and mechanical loads gets controllers out of tune. Ken shows us the math and algorithms behind a self-tuning controller that knows how to maintain efficiency.

Every control engineer is familiar with the problem of tuning the parameters of a digital controller to suit the characteristics of a specific plant. Parameters perfect in one system may be wildly unstable or ineffective in another.

The engineer adjusts the control parameters by trial and error—guided by instinct and experience, sometimes using heuristic rules—until the plant exhibits the desired behavior. The controller is then sent to the plant with the assumption that it will remain reasonably constant.

But, what if that assumption turns out to be false? Many systems have characteristics that change with power supply, mechanical load, or wear on the parts. In these cases, the quality of the control may degrade because the controller is no longer properly tuned.

We need a self-tuning controller that automatically deduces a plant's characteristics and adjusts control parameters to maintain the desired behavior.

The self-tuning control algorithm I present starts with a discrete-time proportional-derivative (PD) control algorithm. Piggybacked on that is a recursive least-squares (RLS) algorithm borrowed from adaptive DSP theory, which estimates the plant's characteristics from its I/O datastreams.

The PD and RLS algorithms are well known and understood. What's not obvious is how to feed the results of the RLS back into the PD.

SYSTEM ANALYSIS

Figure 1 shows a typical system. Signal $d(k)$ is the desired plant output, and $y(k)$ is the actual output. The error signal $x(k)$ is the difference between these outputs. The error signal is usually represented as $e(k)$, but I'm reserving that symbol for something else. The controller output is $u(k)$.

For all signals, I prefer to normalize full scale to 1. That way, the control algorithm can run independent of the application. Other software in the controller can include scaling factors as necessary.

In discrete time, the plant output is described by:

$$y(k+1) = -ay(k) + bu(k) \qquad (1$$

where a and b are the plant parameters. The controller function is described by:

$$x(k) = d(k) - y(k) \qquad (2$$
$$u(k+1) = u(k) + K_p x(k) + K_d [x(k) - x(k-1)]$$

where K_p and K_d are the respective proportional and derivative control parameters.

Assuming the controller can maintain this plant at all, the output $y(k)$ eventually reaches a steady state in response to a step input in $d(k)$, at which time $x(k) = 0$ within some band of tolerance. But, "eventually" doesn't necessarily constitute good, predictable control.

I'd like to optimize the plant's transient response to a step input according to a defined measure by specifying the desired response time and overshoot.

In general, analytical solutions to discrete-time equations aren't possible so to describe the transient response,

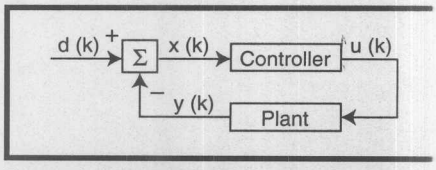

Figure 1—*In a typical feedback control system, the target and actual plant outputs are d(k) and y(k), respectively.*

Listing 1—*This listing shows the combined RLS-PD control algorithm in C.*

```
#include <stdio.h>
#include <stdlib.h>

/* Run-time Signals */
float y, yz1;              /* plant output y(k), y(k - 1) */
float u, uz1;              /* control signal u(k), u(k - 1) */
float d;                   /* desired (target) plant output */
float x, xz1;              /* rate error x(k), x(k-1) */

/* Controller */
float Kp, Kd;              /* PD control parameters */
float M, coeff;            /* intermediate value for Kp and Kd */

/* RLS */
float i1, i2, l;           /* intermediate values for vector g */
float g1, g2;              /* elements of vector g */
float p11, p12, p21, p22;/* elements of matrix P */
#define INIT_P 1000        /* estimate plant parameters, and */
float a, b;                /* elements of vector w */
float alpha;               /* estimation error */
int rls_cntr;              /* RLS iteration counter */
#define RLS_CNT_VAL 10
float get_actual_rate(void); /* called function prototypes */
float get_desired_rate(void);
void set_output(float);
void rls_pd_init(void)
{
  uz1 = yz1 = xz1 = 0;     /* clear run-time signals */
  rls_cntr = RLS_CNT_VAL;/* initialize RLS */
  a = b = 0;
  coeff = 64 / 49;
  Kp = 1;                  /* set arbitrary PD parameters */
  Kd = 1;
}
void rls_pd(void)
{                          /* Eq. 2, Control Output */
  y = get_actual_rate(); /* get run-time data */
  d = get_desired_rate();
  x = d - y;               /* calculate rate error signal */
  u += Kp*x + Kd*(x - xz1); /* calculate control signal */
  if (u > 1.00)            /* bound u to 0 to 100% */
    u = 1.00;
  else if (u < 0)
    u = 0;
  set_output(u);           /* set control output */
  if (rls_cntr == RLS_CNT_VAL) {  /* if time to reinitialize RLS */
    p11 = p22 = INIT_P;
    p12 = p21 = 0;
  }
  if ((--rls_cntr) == 0)  { /* if time to calculate Kp and Kd */
    rls_cntr = RLS_CNT_VAL;
    M = (a + 1) / b;
    Kp = (64/49)*M*(a+1);
    Kd = M / 7;
  }
  i1 = p12*uz1 - p11*yz1;/* Eq. 19, Gain vector g */
  i2 = p22*uz1 - p21*yz1;
  l = 1 + i2*uz1 - i1*yz1;
  g1 = i1 / l;
  g2 = i2 / l;
  alpha = y + a*yz1 - b*uz1; /* Eq. 22, Estimation Error alpha */
  a += alpha*g1;           /* Eq. 21, Estimated Plant Parameters a */
  b += alpha*g2;           /*   and b */
  p11 -= g1*i1;            /* Eq. 20, Inv Correlation Matrix P */
  p12 -= g1*i2;
  p21 -= g2*i1;
  p22 -= g2*i2;
  xz1 = x;                 /* z^(-1), Increment Time */
  yz1 = y;
  uz1 = u;
}
```

I'll convert equations (1) and (2) to continuous time using these approximations:

$$f_d(k) = f_c(k\tau) \tag{3}$$

$$\begin{aligned}\dot{f}_c(t) &\approx \frac{1}{\tau}[f_d(t+\tau) - f_d(t)] \\ &\approx \frac{1}{\tau}[f_d(t) - f_d(t-\tau)]\end{aligned} \tag{4}$$

where f_d is the discrete-time-sampled version of the continuous-time variable f_c, and τ is the sample period.

This approximation is not without risk. If the sample period is too long relative to the system's time constants or rise time, the first-derivative approximation is too inaccurate to be useful.

I developed the algorithm on a system with time constants 10–20 times the sample period. In this case, the magnitude and phase distortion introduced by sampling was negligible.

The next step is to convert the continuous-time versions of equations (1) and (2) to a single equation describing the response to a step input. By subtracting $y(k)$ and dividing by τ, equation (1) can be rearranged as:

$$\begin{aligned}&\frac{1}{\tau}[y(k+1) - y(k)] = \\ &\quad -\left(\frac{a+1}{\tau}\right)y(k) + \frac{b}{\tau}u(k)\end{aligned} \tag{5}$$

Using the continuous-time approximations in equations (3) and (4), equation (5) becomes:

$$\dot{y}(t) = -\left(\frac{a+1}{\tau}\right)y(t) + \frac{b}{\tau}u(t) \tag{6}$$

If we identify:

$$\begin{aligned}\frac{1}{J} &= \frac{b}{\tau} \\ \frac{F}{J} &= \frac{a+1}{\tau}\end{aligned} \tag{7}$$

then equation (6) simplifies to:

$$J\dot{y}(t) = u(t) - Fy(t) \tag{8}$$

which is the familiar equation for a rotating system relating the moment of inertia (J), the coefficient of friction (F), and the driving force (u) to rotational speed (y).

To derive the continuous-time approximation to equation (2), first define the continuous-time proportional control parameter $K_{pc} = K_p/\tau$.

Then, by rearranging and dividing by τ, equation (2) becomes:

$$\frac{1}{\tau}[u(k+1)-u(k)] = K_{pc}\,x(k)+\frac{K_d}{\tau}[x(k)-x(k-1)] \quad (9)$$

Using equations (3) and (4), equation (9) becomes:

$$\dot{u}(t) = K_{pc}\,x(t)+K_d\,\dot{x}(t) \text{ or}$$

$$\dot{u}(t) = K_{pc}d(t)-K_{pc}y(t)+K_d\dot{d}(t)-K_d\dot{y}(t) \quad (10)$$

By differentiating equation (8) and substituting (10), we obtain the second-order differential equation for the combined plant-controller system:

$$J\ddot{y}(t)+(K_d+F)\dot{y}(t)+K_{pc}y(t) = K_d\dot{d}(t)+K_{pc}d(t) \quad (11)$$

The transient response of equation (11) to a step input needs to be analyzed to quantify the response time and maximum overshoot. To analyze the transient response, take $d(t)$ to be a unit step function and assume $y(0) = 0$, $y'(0) = 0$, and $d(0) = 0$.

Then, taking the LaPlace transform of equation (11) and substituting:

$$D(s) = \frac{1}{s}$$

which is the LaPlace transform for a step function of 1, we get:

$$Y(s) = \frac{K_d s + K_{pc}}{s(Js^2+(K_d+F)s+K_{pc})} \quad (12)$$

The interesting characteristics of equation (12) can be gleaned by rewriting it as:

$$Y(s) = \left(\frac{\omega_n^2}{c}\right)\left(\frac{s+c}{s(s^2+2\zeta\omega_n s+\omega_n^2)}\right) \quad (13)$$

where the natural frequency (ω_n), the zero ($-c$), and the damping coefficient (ζ) are defined as:

$$\omega_n = \sqrt{\frac{K_{pc}}{J}}$$
$$c = \frac{K_{pc}}{K_d} \quad (14)$$
$$\zeta = \frac{K_d+F}{2\sqrt{K\ J}}$$

For the controller to be self-tuning, it must know what constitutes being in tune.

The desired behavior of the plant must be defined in terms of a set of mathematical criteria that use data available to the controller.

Three criteria to consider are the type of response (underdamped, critically damped, or overdamped), the damping coefficient, and the maximum overshoot to a step input.

If some overshoot can be tolerated, the quickest convergence of the response to the desired value can be achieved with an underdamped system, meaning the damping coefficient is between 0 and 1.

A damping coefficient close to 0 yields a shorter rise time but greater overshoot. With a damping coefficient close to 1, overshoot is small but the rise time is longer.

Unless there are other system requirements that affect the selection of the damping coefficient, 0.5 is a good compromise between reducing overshoot and shortening the response.

For a given damping coefficient in an underdamped system, one more factor affects the overshoot: the ratio

between the zero ($-c$) and the real part ($-\zeta\omega_n$) of the pair of complex poles [1].

If the ratio:

$$\alpha = \frac{c}{\zeta\omega_n}$$

is close to 1, the overshoot can be as high as 70%. Larger ratios reduce the overshoot, and by the time $\alpha = 4$, the overshoot is down to about 20%.

The minimum possible overshoot is about 14% as α approaches infinite. I used $\alpha = 16$ to keep overshoot below 20% allowing for some noise, but any number greater than 4 is just as good.

By setting the damping coefficient (ζ) to 0.5, $\alpha = 16$ reduces to:

$$\sqrt{K_{pc} J} = 8 K_d$$

From this and the definition of ζ in equation (14), it follows that:

$$K_d = \frac{F}{7}$$

$$K_{pc} = \frac{(64 F^2)}{(49 J)} \qquad (15)$$

$$K_p = \tau \frac{64 F^2}{(49 J)}$$

RLS ALGORITHM

Now that we can calculate K_p and K_d from the plant characteristics J and F, what next? We know the sample period τ, and J and F are functions of a and b, in equation (6).

So, if we can calculate a and b, we are done. To do that, I returned to the domain of discrete-time and borrowed the RLS algorithm from DSP theory.

Least-squares is a method of finding the best solution to an overdetermined set of simultaneous equations. The solution to the least-squares equation is computationally expensive, involving n-dimensional matrix inversion and multiplication, where n is the number of simultaneous equations.

Such a calculation may be beyond the capability of a simple microprocessor. But, there's a recursive solution to the equation in which a simpler set of calculations is performed for each equation individually, and the results from one set of calculations are fed into the next.

To get the recursive solution, I need to cast the problem into a standard form known as the deterministic normal equation. First define the data vector $\boldsymbol{a}$:

$$\mathbf{a}^T(k) = [-y(k-1), u(k-1)] \qquad (16)$$

and the estimated parameter vector $\boldsymbol{w}$:

$$\mathbf{w}^T = [a, b]$$

The performance measure used to judge the estimated parameters is the difference between the measured output $y(k)$ and the predicted output obtained from the estimated parameter vector in equation (1). This estimation error $e(k)$ is:

$$\begin{aligned} e(k) &= y(k) - (-ay(k-1) + bu(k-1)) \\ &= y(k) - \mathbf{a}^T(k)\mathbf{w} \end{aligned}$$

For a datastream of n samples, we get n instances of equation (16), which can be cast into matrix form by defining the data matrix $\boldsymbol{A}$:

$$\mathbf{A}(k) = \begin{bmatrix} \mathbf{a}^T(k) \\ \vdots \\ \mathbf{a}^T(k-n+1) \end{bmatrix} \qquad (17)$$

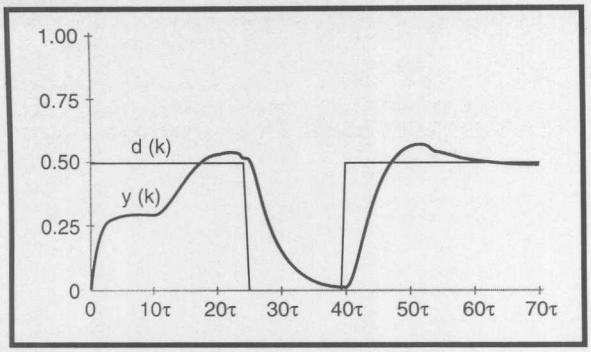

Figure 2—*Control is sluggish for the first ten samples until new control parameters are calculated. At the second step input in d(k), the controller is tuned.*

the error vector ***e***:

$$\mathbf{e}^T(k)=[e(k),\ldots,e(k-n+1)]$$

and the measured data vector ***y***:

$$\mathbf{y}^T(k)=[y(k),\ldots,y(k-n+1)]$$

From these definitions, ***e*** equates to:

$$\mathbf{e}(k)=\mathbf{y}(k)-\mathbf{a}(k)\mathbf{w}$$

In the least-squares method, the square of ***e*** should be minimized. The squared error function is:

$$\begin{aligned}E(k)&=\mathbf{e}^T(k)\mathbf{e}(k)\\&=\mathbf{y}^T\mathbf{y}-\mathbf{y}^T\mathbf{A}\mathbf{w}-\mathbf{w}^T\mathbf{A}^T\mathbf{y}+\mathbf{w}^T\mathbf{A}^T\mathbf{A}\mathbf{w}\end{aligned}$$

The surface defined by this function forms a bowl shape over the *a*-*b* plane. The *a*-*b* coordinates under the lowest point in the bowl define the point of minimum error between the measured output and the estimated output calculated from the estimated *a* and *b*.

At this point, the gradient of the error surface equals zero or:

$$\begin{aligned}\frac{\partial E}{\partial \mathbf{w}}&=-2\mathbf{A}^T\mathbf{y}+2\mathbf{A}^T\mathbf{A}\mathbf{w}\\&=0\end{aligned}$$

which reduces to:

$$\mathbf{A}^T\mathbf{y}=\mathbf{A}^T\mathbf{A}\mathbf{w} \qquad (18)$$

This is the deterministic normal equation for least-squares parameter estimation, and the solution ***w*** provides the necessary *a* and *b*.

Deriving the recursive solution to equation (18) is a rocky trek through linear algebra. But, now that the problem has been made to fit Procrustes-wise into the standard form [2], there's no need to reinvent the wheel. Instead, I'll simply assume that a miracle occurs and jump right to the solution.

First, I need to define one new matrix, two vectors, and a scalar. The gain vector ***g*** is defined as:

$$\mathbf{g}(k)=\frac{\mathbf{P}(k-1)\mathbf{a}(k)}{1+\mathbf{a}^T(k)\mathbf{P}(k-1)\mathbf{a}(k)} \qquad (19)$$

where the inverse correlation matrix ***P*** is:

$$\mathbf{P}(k)=\mathbf{P}(k-1)-\mathbf{g}(k)\mathbf{a}^T(k)\mathbf{P}(k-1) \qquad (20)$$

The estimated parameter vector $\mathbf{w}=[a,b]^T$ is evaluated by:

$$\mathbf{w}(k)=\mathbf{w}(k-1)+\mathbf{g}(k)\alpha(k) \qquad (21)$$

where the estimation error α is:

$$\begin{aligned}\alpha(k)&=y(k)-\mathbf{a}^T(k)\mathbf{w}(k-1)\\&=y(k)-\mathbf{w}^T(k-1)\mathbf{a}(k)\end{aligned} \qquad (22)$$

The RLS parameter estimation algorithm proceeds as follows. First, equation (19) calculates the gain vector ***g***, and equation (22) calculates the estimation error (α).

Using ***g*** and α, the estimated parameter vector ***w*** is recursively updated in equation (21). In the last step, equation (20) recursively updates ***P***. The algorithm is initialized at $k = 0$ with $\mathbf{w}(0)=\mathbf{0}$, and $\mathbf{P}(0)=p\mathbf{I}$, where *p* is an arbitrary number greater than 1.

Once *a* and *b* are estimated, the control parameters K_p and K_d are found from equations (7) and (15), which reduce to:

$$K_d=\frac{(a+1)}{7b} \qquad (2.$$

$$K_p=\left(\frac{64}{49}\right)\frac{(a+1)^2}{b}$$

In equation (23), the control parameters seem independent of the sample rate, and in a sense they are. The sample rate doesn't directly contribute to the control parameters, but a different sample rate would be made manifest by different plant parameters *a* and *b*.

PD_RLS CONTROL ALGORITHM

Several issues need to be considered before the RLS algorithm is ready. One is the value of *p* used to initialize $\boldsymbol{P}(0)$.

The only constraint on the value of *p*—that the correlation matrix $\boldsymbol{P}^{-1}(k)$ has to stay invertible—is met by setting *p* to a number greater than 1 [3]. Otherwise, *p*'s value has little effect on the results, except that a larger number results in faster convergence.

Another consideration is when to recalculate the control parameters K_p and K_d. The estimated plant parameter vector ***w*** is inaccurate for a short time after initialization. Control parameters calculated from immature parameter estimates don't result in good control.

In a noisy system with a constant target rate, the system reaches a steady state with the error signal equal to zero plus a noise component. In this case, the RLS algorithm can get confused and calculate plant parameters that reflect the noise more than the true plant characteristics.

You could skip recalculating the estimated plant parameters unless the target rate changes by an amount sig-

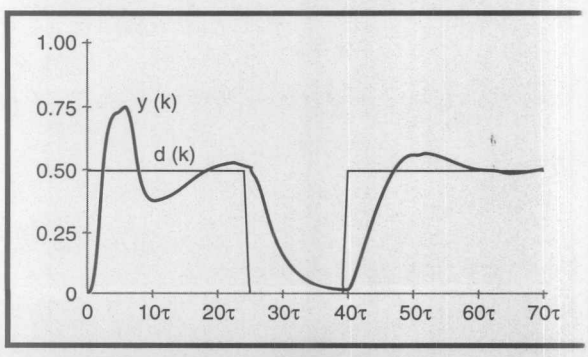

Figure 3—*At first, the output wildly overshoots. Again, the controller is tuned at the second step input.*

nificantly greater than the magnitude of the noise.

I found empirically that I can obtain good results by running ten iterations of the RLS algorithm before using the estimated plant parameters to recalculate the control parameters. Then, I reinitialize the RLS algorithm and recalculate the control parameters after every ten iterations to keep up with changing plant characteristics.

Listing 1 shows the combined PD-RLS control algorithm in C. The intermediate vector $\boldsymbol{i}$ used in calculating $\boldsymbol{g}$ is also used in the calculation of $\boldsymbol{P}$. This works because $\boldsymbol{i} = \boldsymbol{PA}$ in (19), so $\boldsymbol{i}^T = \boldsymbol{A}^T\boldsymbol{P}$, which is what (20) needs.

SYSTEM SUCCESS

Figures 2 and 3 show data obtained from trial runs on the same system. In each case, the target rate is 0.5 from time 0 to time 2.5 s, after which it goes to 0. At 4 s, it returns to 0.5 again.

In Figure 2, the control parameters start at $K_p = 0.001$ and $K_d = 1$. The result is a sluggish response for the first second until the control parameters are recalculated. The response to the input pulse at 4 s reaches the target rate within 1 s, overshoots by about 10%, and then settles down.

In Figure 3, the control parameters start at $K_p = 1$ and $K_d = 0.001$, which results in almost 50% overshoot in the first second. But again, the response to the input pulse at 4 s is well-behaved.

COMPUTERS IN THE FIELD?

This algorithm allows the controller to calculate automatically the parameters needed to achieve the desired response. It produces a transient response with specific characteristics, but other characteristics can be chosen. Different combinations of the damping coefficient and the zero in equation (13) simply change the constants used in equation (20).

I implemented this algorithm on a 68HC11 controller with a 12-MHz clock. I was able to use a sample period of 100 ms with plenty of time available for other system functions.

The controller was used on a fertilizer spreader—literally, "in the field"—in which the fertilizer application rate varied with the speed of the truck through the field to apply a constant amount of fertilizer per unit area.

To complicate the issue, the machinery's response changed with power-supply voltage, load, and usage as bearings and gears jammed with fertilizer and dust. Also, the operators wanted to use the same controller on several different systems and to switch from one to another at any time without retuning the controller.

The algorithm proved to be robust. The controller could connect to a new system and be tuned in 1–2 s. And, the tuning changed with plant characteristics, keeping response consistent.

Ken Baker works as a senior design engineer at Guidant/CPI, a medical-device company. You may reach Ken at qc03660@stp03.guidant.com.

SOFTWARE

The algorithm in 68HC11 assembly code, available on the Circuit Cellar BBS, makes use of floating point functions available on the Internet at http://freeware.aus.sps.mot.com:80/freeweb/amcu_ndx.html#mcu11. HC11FP11.Asm is freeware maintained by Motorola.

REFERENCES

[1] K. Ogata, *Modern Control Engineering*, Prentice Hall, Englewood Cliffs, NJ, 243, 1970.

[2] S. Haykin, *Adaptive Filter Theory*, Prentice Hall, Englewood Cliffs, NJ, 381, 1986.

[3] R. Iserman, *Digital Control Systems*, Springer-Verlag, Berlin, 367, 1989.

SOURCES

68HC11
Motorola
MCU Information Line
P.O. Box 13026
Austin, TX 78711-3026
(512) 328-2268
Fax: (512) 891-4465

IRS

413 Very Useful
414 Moderately Useful
415 Not Useful